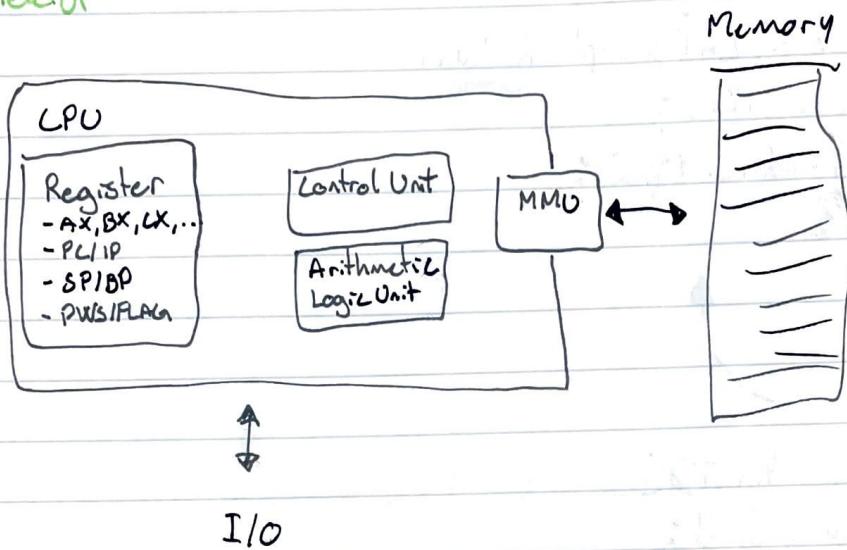


Repetisjon - Kompendie

1. Intro - datamaskinarkitektur

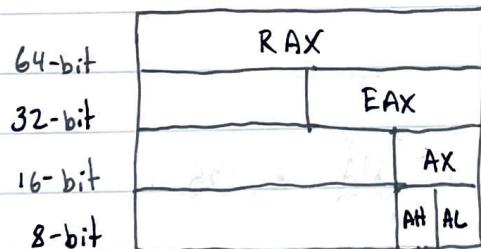
= ord/
begreper

NB!-Gjør alle "KEY PROBLEM"
Arkitektur



von Neumann architecture
Shared busses

Harvard/modified Harvard architecture
data and instruction buss



IP - Instruction pointer
SP - Stack pointer
BP - Base pointer

ISA - Instruction set architecture
Firmware - Software som er innebygget.

Instruksjoner

- Move/Copy data `mov`
- Math functions `add`, `sub`
- Func related `call`, `ret`
- Jumping `jmp`, `je` (jump if equal), `jne` (jump not equal)
- Comparing `cmp`
- Stack `push`/`pop`

CPU-ens funksjon

```
while (not HALT){  
    IR = mem[PC]; // Instruction Register  
    PC++;  
    execute(IR);  
    if (IRQ){ #Interrupt Request  
        save PC();  
        load PC(IRQ);  
    }  
}
```



```
while (not HALT){  
    IR = mem[PC];  
    PC++;  
    execute(IR);  
    if (IRQ)
```

Register vs. Fysisk minne (RAM)

Register tilhører prosessen som kjører NÅ og OS



Stacken inneholder:

- Lokale variabler
- Funksjons-argumenter
- Return-addresser

Hver funksjon har egen større frame

1.2. Software

Compiling

```
int main() {  
    int i = 10;  
    if (i == 3){  
        ...  
        ...  
        ...
```

gcc
→

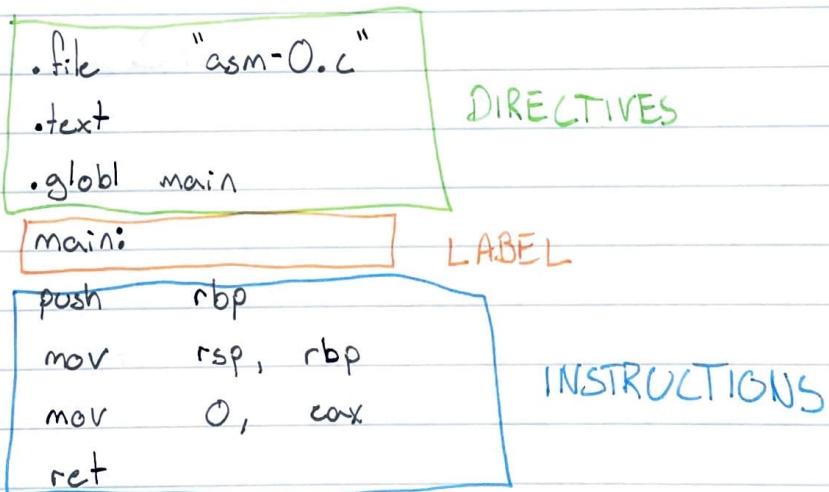
1000110100011101
0110011001111110
100.....

32 vs. 64 bit

Samme C-kode, annen assembly.

gcc -S asm-0.c # 64-bit på 64-bit OS

gcc -S -m32 asm-0.c # 32-bit



mnemonic - instruksjon (mov)

mnemonic suffix - b (byte), w (word), l (long), q (quadword)

operand - is an argument

operand prefix - % is a register, \$ is a constant

address calculation movl - 4(%ebp), %eax

CPU-terms

- Clock - speed/rate
- Pipeline, superscalar - flere funksjonelle enheter følger flere arittmetiske enheter.
- Micro operations $\xleftarrow{\text{into}}$ Machine instructions
- Out-of-Order execution - instruksjoner utføres ikke alltid i rekkefølge

Hyperthreading

En prosessorkjerne kan ha flere threads/prosesser.

Cache = Smart

Write-through - cache, then straight to memory

Write-back - write to cache, mark dirty shutdown = RIP

2. Operativsystemer og prosesser

Intro

Hva gjør operativsystemet?

EXAMEN!
SPØRSMALE

Det virtualiserer fysiske ressurser slik at de blir brukervennlige.

Det styrer bruk av ressurser på datamaskinen.

Virtualisering

CPU og minne virtualiseres slik at hvert program tror det har en hel data maskin for seg selv.

Concurrency (samtidighet)

Flera ting kan skje samtidig

Persistanse (varighet)

Permanent lagring med I/O

Design mål

Virtualisering - lage abstraksjoner

Performance - minimize overhead (minimere unødvendig bruk av ressurser)

Security - protect / isolate applications

Reliability - stabilitet

Energy-efficient - miljøvennlig

Prossesser

Policy and mechanism is separated.

Prosses - kjørende kode. Kjørende instans av program.

Program - Kode skrevet for å gjøre noe.

States

Ready

Running

Blocked

Bruk av CPU

t	P1	P2	
1	Running	Ready	
2	Running	Ready	
3	Running	Ready	
4	Running	Ready	I/O-done
5	-	Running	
6	-	Running	
7	-	Running	
8	-	Running	P1-done

PCB - Process Control Block

Lager all info om en prosess.

Er en datastruktur.

Inneholder: PID, state, memory, åpne filer

Prosess egenskaper (Characteristics)

CPU-bound mye CPU-bruk, maskinering, multimedia, husk at hyperthreading IKKE hjelper en CPU-intensiv prosess.

I/O-bound vent mye på I/O

Memory-bound mye minne, ofte også CPU-bound

Real-time deadlines, soft real-time (multimedia), hard real-time (robotics).

Batch vs. Interactive batch har ikke I/O

Service "kjører uten en bruker logget inn"

3. System calls

3.1 System kall

fork();

int rc = fork();

Parent

Child

rc = Child PID rc = 0

fork() lager ~~ha~~ en kopi av den kjørende prosessen. Den kalles en "Child process".

Rekkefølgen disse kjører i er IKKE deterministisk.

Start sett vil parent kjøre først, men ikke alltid.

fork() bruker copy on write for å unngå uavhengig alloksering av minne. Dette kan skape problemer som vi skal løse i senere kapitler.

exec();

Dette systemkallet tar den kjørende prosessen, og "gjør den en" til et annet program/prosess.

HVORFOR?

utan fork(); exec(new_program);

og ikke:

CreateProcess(new_program); # Brukes i Windows

4

BETYKNINGER

"Separasjon av fork() og exec() gjør det mulig å endre omgivelser og parametre etter fork() og før exec()."

Signals

Signalerer til prosesser
eks:

kill proses1

3.2 Process execution

Direct execution

OS

Create entry for process-list

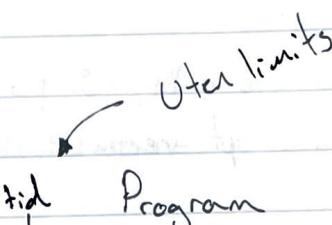
Allocate memory for program

Load program into memory

Set up stack argc/argv

Clear registers

Execute call main



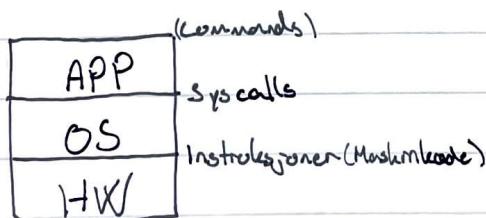
Free memory of process

Remove process from list

Direct execution - run the process directly on the CPU.

Restricted operation

"Hjem kan hva?"



APPER bruker syscalls for å gjøre ting de ikke har lov til. Da må de først gjennom OS-et.

Nærmord

user mode (applikasjonen)

kernel mode (operativsystemet)

mode switch (mellom user og kernel mode)

context switch (mellom prosesser)

Trap-instruksjon

Før å bruke sys-calls, må programmet utføre en trap-instruksjon.

Tilsvarende "interrupt vector table" fra datavark

Når kjører Os-ct?

- (Trap) Software interrupt/Syscalls (synchronous)
- (Trap) Exception (synchronous)
- Hardware interrupt (asynchronous)

Timer interrupt

4. Scheduling

4.1 Turnaround time

Antagelser:

1. Hver jobb kjører like lenge
2. Alle jobber ankommer samtidig
3. Når startet, så kjører jobben ferdig
4. Bruke bare CPU
5. Vi kjenner kjøretiden

$$T = T_{\text{fordig}} - T_{\text{ankomst}}$$

Turnaround time - tiden fra en prosess kommer inn i systemet og går ut igjen.

FIFO

First In, First Out, Her droppes 1.

Convoy effect:



En lang jobb kan blokkere korte jobber.

SJF

Shortest Job First, Her droppes 2.

Non-preemptive - Den kjører alltid prosesser ferdig.
Her kan korte jobber ende opp med å blokkere lange jobber som aldri får CPU-tid.

STCF

Shortest Time to Completion First, Her droppes 3.

~~Non-preemptive~~ Preemptive - Den kan avslutte en jobb før den er ferdig

Bra turnaround-time. Dårlig response time.

4.2 Response time

Tiden fra en prosess kommer inn i systemet til den kjører første gang.

Merk: ~~Antall tidsrommer~~

$$T = T_{\text{først kjørt}} - T_{\text{ankomst}}$$

Round Robin



- Deler i time-slices
- Context switch kostar tid

Overlap

Når en prosess venter på I/O, kan en annen kjøre

4.-5. gjelder ikke lenger.

Aldri i CPU-en gjøre ingenting.

4.3 MLFQ (Multi Level Feedback Queue)

Basics

1. $\text{Pri}(A) > \text{Pri}(B) \rightarrow A$ -kjøren
2. $\text{Pri}(A) = \text{Pri}(B) \rightarrow A$ og B kjører Round Robin

Prioritet

3. Når en jobb kommer inn, så har den høyeste prioritet
- 4a. Hvis en jobb bruker hele sin time-slice mens den kjører, så går den ned et prioritetsnivå.
- 4b. Hvis den gir opp CPU før timeslisen er over, blir den på samme prioritet.

Boost

4. Når en prosess bruker opp tiden sin på et nivå, så går den ned.
5. Etter en tidsperiode S , går alle til øverste prioritet.

4.4 Lik Fordeling

Tilfeldig

Før at alle prosesser skal kunne slippe til, så kan man velge tilfeldig hvem som får tilgang til CPU-en.

4.5 Fler prosessorer (CPU-en)

Fysikk setter begrensninger for hvor raskt prosessoren kan være. Raskere prosessoren produserer også mer varme, og kan være vanskelig å kjøle ned.

I stedet for raskere prosessorer, så kan vi prøve fler prosessorer.

Affinity

Det er en fordel å la samme proses alltid kjøre på samme CPU. Derfor prøver **affinity scheduling** i oppgårdette.

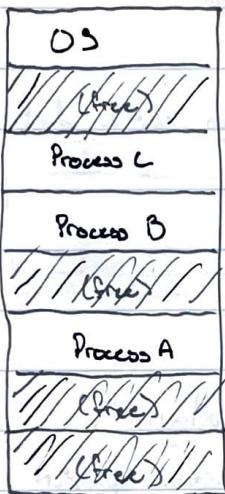
Grang scheduling

Gir det mening at tråden fra samme prosess kjører på samme CPU?

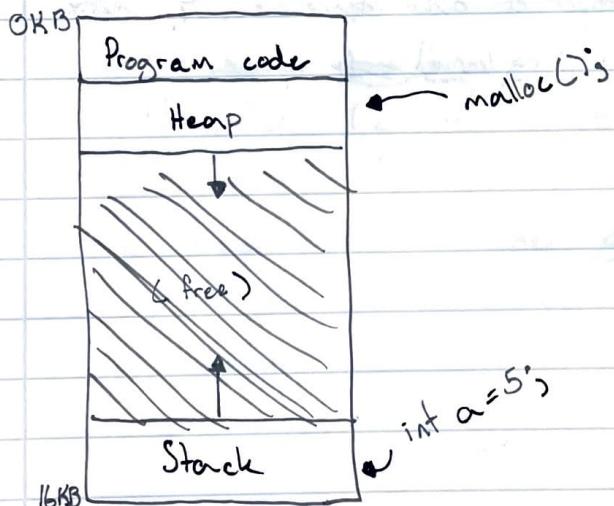
5. Address Spaces and Address Translation

5.1 Address Space

Multiprogramming



Address space



Minnet er virtualisert fordi programmet ikke er lastet i minne der den tror den er.

Mal

Gjennomsiktighet: Virtualiseringen skjer "behind the scenes", og skal ikke merkes.

Effektivitet: 1 tidlig øgrom

Beskyttelse: Isolert fra andre addresserom.

5.2 Minne API

int x; → stack

int *x = (int *) malloc(sizeof(int));

→ stack → heap

globale variabler er i Data-segment, IKKE heap.

→ malloc() returnerer en void pointer, så man kan caste det til hva man ønsker.

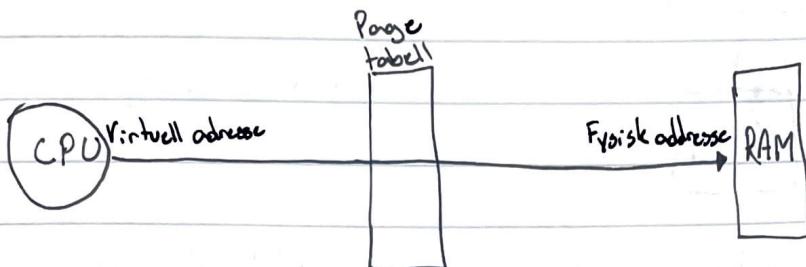
Free memory

- free(x);
- Det er lett å gjøre feil. Bruk verktøy som valgrind.
- "use after free", et angrep en sårbarhet som oppstår når i pekeren fortsatt peker til stedet i minnet etter free()
- Förste heksadesimal i adresser er alltid større enn 7, dette er også delingen mellom user og kernel ~~space~~ space

5.3 Adresse oversetting

Address space and address translation

int k = 4;
⇒ mov \$4, -4(%ebp)



- Linux og Windows bruker stort sett 48-bits addresser

Multiprogrammering - flere programmer i minne

Hvorfor virtualisere minne?

- Isolere prosesser
- Fasst minneområde å forholde seg til.

Heap vs. stack

- I heap må du allokerere plass manuelt
- Stack går automatisk
- Heap har mye mer plass
- I heap må du frigjøre minne også

NB!

- Ingen addresser starter høyere en 0x7... Fordi halvparten av minnet er sett av til kernel-mode. Dersom man prøver å tildele plass i kernel-området, så får man segmentation fault

Segmentert minne

Funker for 32-bit

Funker ikke på x86 64-bit systemer

Free space management

- Freelist 2, 3, $\overbrace{7, 8, 9, 10, 11}^{7, 5}$
- Bitmap (bare i kompendiet)
16-bit (0=ledig, 1=oppatt)

8GB ram, 4kB deler

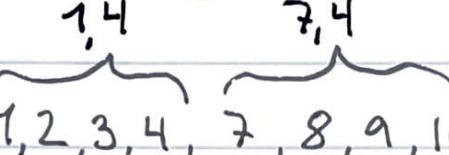
Hvor start bitmap?

$$\frac{8\text{GB}}{4\text{kB}} = \frac{2^{30+3}}{2^{10+2}} = 2^{21} \text{bit} = 2^{18} \text{Byte}$$

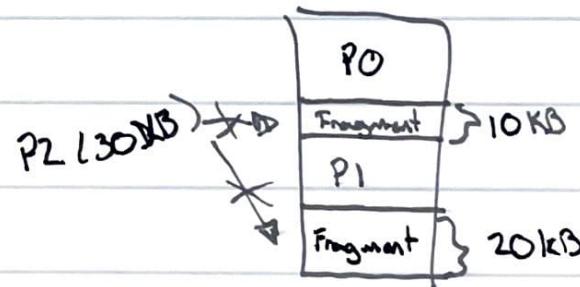
$= 256\text{kB}$



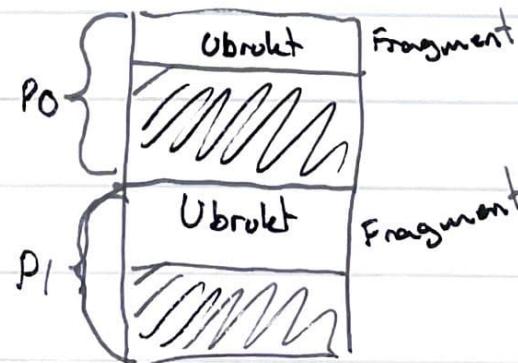
5.5 Free space management

- Bitmap 
- Free list 1,2,3,4,7,8,9,10

Ekstern fragmentering - Når det er ledig plass mellom addresserom som kan hindre å få plass til noe nytt.



Intern fragmentering - En prosess får mer plass enn den trenger



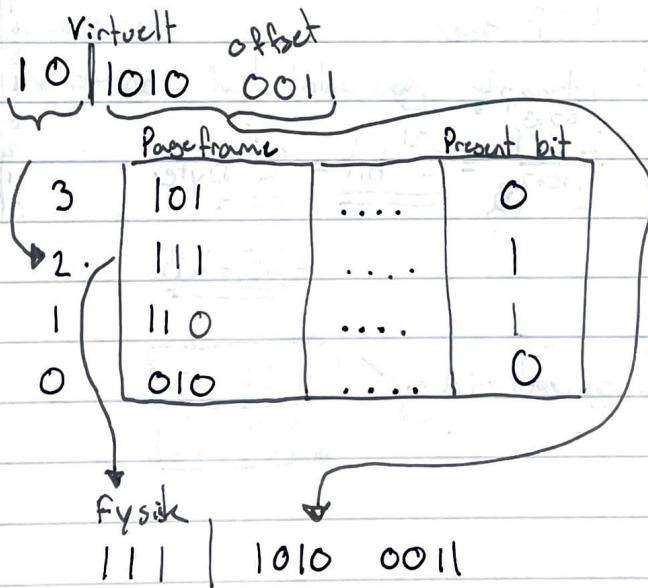
Paging (kap 18, se også figure i kompendie)

- Linux og Windows bruker stort sett 4kB pages
 - MMU i CPU før det til å se ut som prosesser har sitt eget store minne

Address translation (Oppgaver på eksamen)

2 Mb - page størrelse \rightarrow 21 bit for adresse i page

Eks: 8-bit page



Hva inneholder page tabell?

- Present bit
 - Protection bits
 - Reference bit
 - Dirty bit
 - Caching bits

6. Memory management

6.1 Faster translation

1. TLB-cache (Translation Lookaside Buffer)

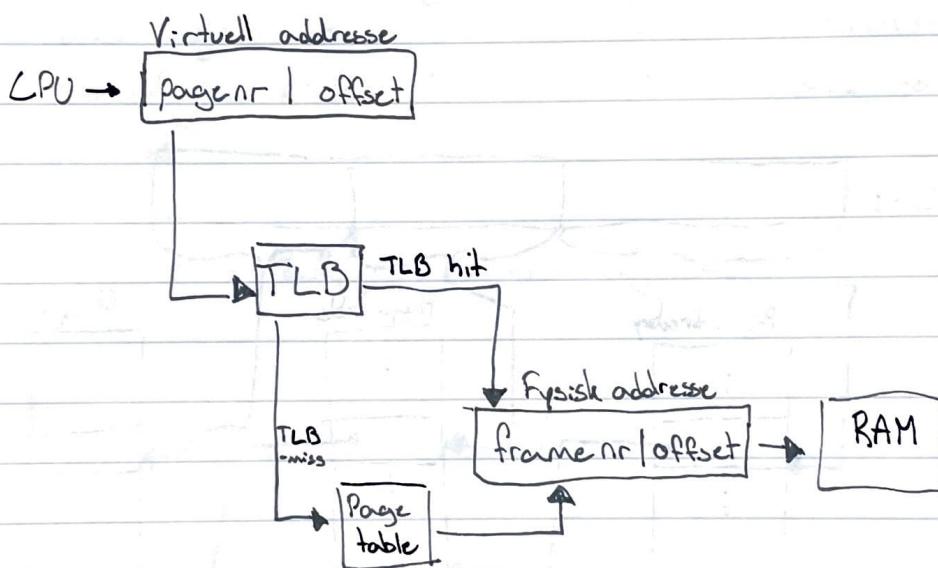
2. Page table far mye plass

- Mult-level page table

- Inverted page table

TLB

Er en CPU-cache



- 70% hit rate

- Hvorfor cache?

- Spatial locality

- Temporal locality

- OS behandler TLB → RISC

- HW behandler TLB → CISC

ASID

Hva er en TLB-entry?

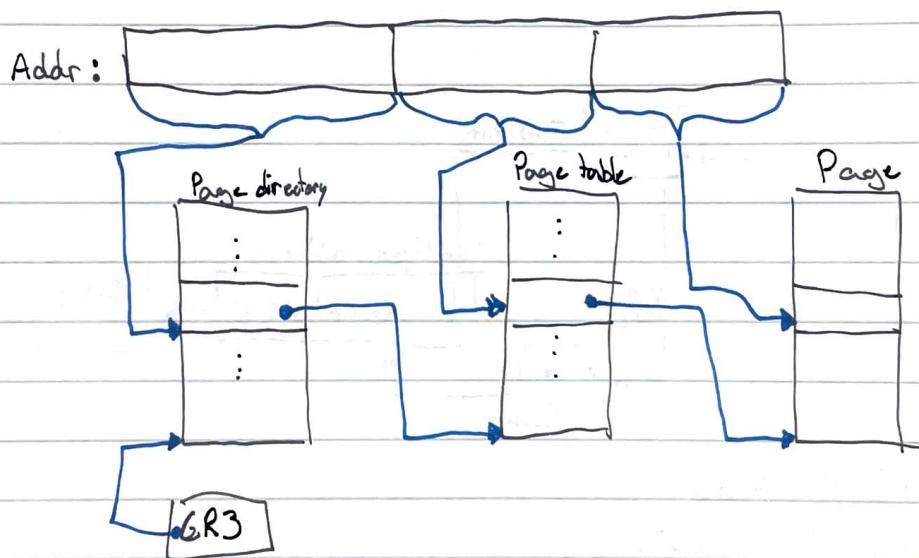
- En kopi av Page Table Entry
- ASID - Address Space Identifier gjør at du ikke må flusher TLB ved hver contextswitch.

6.2 Mindre Page Tables

Større pages?

- Større pages betyr farre pages, og kortere PageTable.
- X86 støtter 4KB, 2MB og 1GB
- Gir mer intern fragmentering

Multi-level PT



Inverted Page Table

- En entry per fysisk page

6.3 Memory Management

Swap Space

Område på disk som kan brukes dersom fysisk minne er fullt.

Page Fault

Mapping til en page finnes ikke

TLB miss / Soft miss PageTableEntry not in TLB

Minor Page Fault / Soft miss Page in memory, but not marked as present in PTE (Brakt i minne av en annen prosess).

Major Page Fault / Hard miss Page not in memory, I/O required.

6.4 Page Replacement Policies

Parallelle problemer

- "Alt" er bare en cache til noe tregere.
- RAM → Disk → RAM på disk ... osv...

Policies

Optimal - Furthest in the Future. Ikke mulig å bruke.

Det er nyttig å vite hvilken optimale metoden er. På den måten, så kan vi male andre policies mot den.

FIFO - First In First Out

Random - Jon

Least Recently Used (LRU)

Alle metodene funker forskjellig : forskjellige tilfeller.

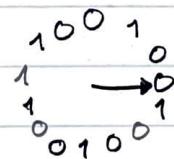
Workloads

No-locality → Alt går litt

80-20 → LRU funker bedre

Looping → Rand funker bedre

Klokke-metode



if 0:

kost ut()

if 1:

= 0

if page-used:

= 1

Terminologi

Demand paging vs Pre-fetching/ pre-paging

Working set

Thrashing - Når alltid brukes på paging

6.5 Linux

- Kernel logical minne er fysisk bundet, ikke virtuelt minne.
- Multilevel paging
- Huge pages or store pages
- 2Q Cache management
- Sikkerhet
 - NX-bit (kan ikke eksekvere kode fra stacken)
 - ASLR Address Space Layout Randomization
 - Meltdown and spectre

7. Threads and Locks

7.1 Intro

Multithreading

- Et program med en tråd er single-threaded
- PCB vs TCB (Thread Control Block)
- Thread har egen:
 - Stack
 - IPI/PL
 - State
 - Register
- "Miniprossesser" inni en prosess, men deler addresserom.

Hva for threads?

- Brukes for parallelt samarbeid.
- Threads gir høy performance
 - Parallelitet: Bruk alle CPU-lykene
 - Overlap: Ved I/O kan andre tråder slippe til.

API

Godte notater på thread API er tidligere i boka.

Terminologi:

Atomicity - en instruksjon som ikke kan avbrytes midt i. Har enten oppført eller ikke oppført.

Critical section - en sektor som inneholder delte ressurser

Race condition - Når utfallet av et program kan variere basert på når prosessene/trådene kjører; forhold til hverandre.

Deterministisk

Mutual exclusion - Sørge for at bare en gangen har tilgang til kritisk sektor.

Threads Thread API (C-kode, komplett)

- `pthread_create(pthread_t *thread,
const pthread_attr_t *attr,
void *(*start_routine)(void*),
void *arg);`

Pointer til et `pthread_t`-struct vi bruker til å interakte med threaden

Ør kan lage attributter med `pthread_attr_init()`: Vanligvis NULL

Hvilken funksjon skal threaden kjøre? (function pointer) returnerer en void pointer slik at vi kan caste den til hva vi vil.

Argumentene sendt til funksjonen

- `pthread_join(pthread_t thread, void **value_ptr);`

Spesifiserer hvilken thread

Pointer til return-verdi; er void for typecasting